

Research - Kasteran* — Type Theory for Programming Languages

Alpasan, Lois-Kleinner

2026

Abstract

Type theory provides the mathematical foundation for programming language type systems, governing how values are classified and how operations are validated for correctness. This document surveys the major traditions in type theory—structural vs. nominal typing, Hindley-Milner inference, System F polymorphism, and dependent types—and examines how Kasteran* integrates these traditions into a unified type system with linearity, capabilities, and compile-time verification.

Kasteran* — Type Theory for Programming Languages

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Type theory provides the mathematical foundation for programming language type systems, governing how values are classified and how operations are validated for correctness. This document surveys the major traditions in type theory—structural vs. nominal typing, Hindley-Milner inference, System F polymorphism, and dependent types—and examines how Kasteran* integrates these traditions into a unified type system with linearity, capabilities, and compile-time verification.

1. Introduction

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute (Pierce 1). Type systems serve multiple purposes: documentation (types communicate programmer intent), correctness (types prevent invalid operations), optimization (types inform code generation), and security (types enforce access control). Kasteran*'s type system draws on multiple traditions to provide a flexible, expressive, and safe programming environment.

2. Historical Background

The modern understanding of type systems begins with Bertrand Russell's theory of types, developed to resolve paradoxes in set theory (Russell 1). Russell's ramified type theory classified sets into a hierarchy, preventing self-reference and the Liar paradox. Alonzo Church's simply typed lambda calculus formalized functional abstraction with type annotations, establishing the foundation for typed programming languages (Church 1).

The Hindley-Milner type inference algorithm, independently discovered by Hindley and Milner, demonstrated that types for a substantial fragment of functional programming could be inferred

automatically without type annotations (Hindley 1; Milner 1). The algorithm is based on unification: type variables are solved by equating type expressions and finding the most general unifier. Damas and Milner proved that the algorithm infers the principal (most general) type for any well-typed expression (Damas and Milner 1).

Reynolds’ work on parametric polymorphism established the theoretical foundations for generic programming (Reynolds 1). His abstraction theorem (later formalized as parametricity by Wadler) states that polymorphic functions behave uniformly across all type instantiations, guaranteeing that a function of type $\text{Id} \rightarrow \text{Id}$ must be the identity function (Wadler 1). This result provides strong guarantees for generic code.

Girard’s System F extended the simply typed lambda calculus with universal quantification over types, enabling fully parametric polymorphism (Girard 1). System F is the theoretical basis for generics in Java, C#, Rust, and Haskell. The calculus is extremely expressive—all recursive types, encodings of data structures, and abstract data types can be represented within it—but type inference for full System F is undecidable, motivating practical restrictions in programming language implementations.

Dependent types, where types can depend on values, were pioneered by the AUTOMATH project (de Bruijn 1) and later developed in the Calculus of Constructions (Coquand and Huet 1). In a dependently typed language, the type of a function’s return value can depend on the value of its argument, enabling the type system to express precise specifications such as “the function returns a vector of length $n + m$.”

3. Technical Analysis

Kasteran* employs a gradual type system with three levels of typing rigor. In “dynamic” mode, types are checked at runtime. In “static” mode, all types are checked at compile time. In “verified” mode, the type system generates proof obligations for the SMT solver.

The type system is structurally typed by default: two types are considered equivalent if they have the same structure (same fields with same types), regardless of their declared names. This enables flexible, duck-typing-like patterns while maintaining static type safety. Structural typing is formalized through a subtyping relation:

$S <: T$ iff S has at least the fields of T , with compatible types

If a record type T has fields $\{a: \text{Int}, b: \text{String}\}$, any record type S with those fields (and possibly more) is a subtype of T .

Nominal typing is available through the `type` declaration, which introduces a named, opaque type that is distinct from any other type. Nominal types are used to enforce abstraction boundaries and prevent unintended type equivalence.

Polymorphism follows the System F tradition with explicit quantification:

```
fn identity<T>(x: T) → T { x }
```

Type inference recovers Hindley-Milner for the Hindley-Milner subset (rank-1 polymorphism) and uses bidirectional inference for higher-rank types. The inference algorithm proceeds in two phases: (1) constraint generation, which traverses the AST and generates equality and subtyping constraints, and (2) constraint solving, which uses unification to find a satisfying substitution.

GADTs (Generalized Algebraic Data Types) extend the expressiveness of pattern matching by allowing the return type of a constructor to vary:

```
type Expr[T] =
  | Lit(Int)      → Expr[Int]
  | Add(Expr[Int], Expr[Int]) → Expr[Int]
  | Eq(Expr[Int], Expr[Int]) → Expr[Bool]
  | If[T](Expr[Bool], Expr[T], Expr[T]) → Expr[T]
```

GADTs enable type-safe interpreters where the type system tracks the type of each expression. The pattern matching exhaustiveness checker uses GADT type refinement to determine which branches are reachable.

4. Current State of the Art

Modern type systems increasingly incorporate dependent types and theorem proving capabilities. Idris, Agda, and Lean are full dependently typed languages where programs and proofs are unified. The Lean theorem prover and programming language, developed at Microsoft Research, has gained significant adoption in mathematics and formal verification.

Rust’s trait system, while not dependently typed in the full sense, supports “associated type families” and “const generics” that bring type-level computation to systems programming. Rust’s type system continues to evolve, with generic associated types (GATs) and specialization adding expressiveness.

Haskell’s type system extensions—including TypeFamilies, DataKinds, PolyKinds, and TypeInType—have made Haskell one of the most expressive practical type systems. The Glasgow Haskell Compiler (GHC) implements a variant of System F with type-level programming capabilities approaching those of dependent types.

Scala 3 (Dotty) introduces a new type system based on the DOT (Dependent Object Types) calculus, which unifies nominal and structural typing, path-dependent types, and intersection/union types within a principled theoretical framework (Odersky et al. 1).

5. Relevance to Kasteran*

Kasteran*’s type system is designed for practical expressiveness while maintaining decidability and performance. The system is bidirectional: top-level declarations are checked (expected type provided), while local expressions are inferred (type synthesized from context). This design provides the ergonomics of type inference with the predictability of type checking.

The capability system is integrated into the type system as a kind annotation: every type is classified not only by its constructor (Int, String, Array) but also by its capability (linear, affine, unrestricted, borrow). This enables the type checker to verify resource management, memory safety, and concurrency properties alongside traditional type correctness.

Type-level computation is available through a restricted form of dependent types: type functions (type-level functions evaluated at compile time) and const generics (type parameters that are compile-time constants). These features enable expressing complex invariants—array bounds, matrix dimensions, state machine transitions—within the type system.

6. Future Directions

The unification of type theory and program verification remains the grand challenge of type system design. Full dependent types are undecidable for general-purpose programming; partial dependent types or refinement types offer a practical compromise. Liquid types, which combine Hindley-Milner inference with SMT-based refinement type checking, demonstrate that significant verification power can be achieved without requiring full dependent type annotations (Rondon et al. 1).

Another frontier is the integration of effects into type systems. Algebraic effects and handlers require fine-grained effect tracking, which is naturally expressed through the type system’s kind or effect annotation. Kasteran’s capability system provides a foundation for this integration.

Works Cited

- Church, Alonzo. “A Formulation of the Simple Theory of Types.” *The Journal of Symbolic Logic*, vol. 5, no. 2, 1940, pp. 56-68.
- Coquand, Thierry, and Gérard Huet. “The Calculus of Constructions.” *Information and Computation*, vol. 76, no. 2-3, 1988, pp. 95-120.
- Damas, Luis, and Robin Milner. “Principal Type-Schemes for Functional Programs.” *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1982, pp. 207-212.
- de Bruijn, Nicolaas G. “A Survey of the Project AUTOMATH.” *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980, pp. 579-606.
- Girard, Jean-Yves. “Interpretation Fonctionnelle et Élimination des Coupures de l’Arithmétique d’Ordre Supérieur.” *Doctoral Thesis, Université Paris VII*, 1972.
- Hindley, J. Roger. “The Principal Type-Scheme of an Object in Combinatory Logic.” *Transactions of the American Mathematical Society*, vol. 146, 1969, pp. 29-60.
- Milner, Robin. “A Theory of Type Polymorphism in Programming.” *Journal of Computer and System Sciences*, vol. 17, no. 3, 1978, pp. 348-375.
- Odersky, Martin, et al. “The DOT Calculus: A Functional Calculus with Dependent Object Types.” *Proceedings of the 2010 Workshop on Foundations of Object-Oriented Languages*, 2010, pp. 1-10.
- Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.
- Reynolds, John C. “Towards a Theory of Type Structure.” *Programming Symposium*, Springer, 1974, pp. 408-425.
- Rondon, Patrick M., et al. “Liquid Types.” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 159-169.
- Russell, Bertrand. “Mathematical Logic as Based on the Theory of Types.” *American Journal of Mathematics*, vol. 30, no. 3, 1908, pp. 222-262.
- Wadler, Philip. “Theorems for Free!” *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, 1989, pp. 347-359.
- Cardelli, Luca. “Type Systems.” *ACM Computing Surveys*, vol. 28, no. 1, 1996, pp. 263-264.
- Pierce, Benjamin C. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

Harper, Robert. *Practical Foundations for Programming Languages*. 2nd ed., Cambridge University Press, 2016.

Mitchell, John C. *Foundations for Programming Languages*. MIT Press, 1996.

Vytiniotis, Dimitrios, et al. “FPH: First-Class Polymorphism for Haskell.” *Proceedings of the 2006 ACM SIGPLAN International Conference on Functional Programming*, 2006, pp. 1-12.

Sulzmann, Martin, et al. “A Theory of Qualified Types.” *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 1-12.

Leijen, Daan. “Type-Directed Compilation of Algebraic Effect Handlers.” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2020, pp. 1-27.

Simonet, Vincent, and François Pottier. “Constraint-Based Type Inference for Guarded Algebraic Data Types.” *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, 2003, pp. 1-12.

Schrijvers, Tom, et al. “Type Checking with Open Type Functions.” *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming*, 2008, pp. 1-12.

Weirich, Stephanie, et al. “A Specification for Dependent Types in Haskell.” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019, pp. 1-29.